Cocoa API Techniques

Session 302

Cocoa API Techniques

Ali Ozer
Manager, Cocoa Frameworks

What You'll See Today

- Techniques for Cocoa API design
- Conventions used in Cocoa APIs
- Case studies

Topics

- Naming and API conventions
- Object ownership
- Mutability
- Subclassing
- Plug-in design
- Performance of APIs

Naming and API Conventions

Naming Conventions

- Functions
  NSMakeRange( ), NSSetFocusRingStyle()
- Classes
  NSString, NSTableView, ABRecord, DRFolder
- Methods
  selectedTextColor, insertObject: atIndex:
- Enums
  NSWindowBelow, NSWindowAbove

Naming Conventions

- Use prefixes on class names

  NSString, NSImage, NSPreferencePane
  ABPerson, ABRecord
  DRFolder, DRDevice

- Protect against collisions
- Differentiate functional areas

Naming Conventions

- It’s better to be clear than brief

Use

```
insertObject: atIndex:
```

instead of

```
insert: at:
```

or

```
insert::
```

Naming Conventions

- It's better to be clear than brief

  Use
  `removeObjectAtIndex:`
  instead of
  `removeObject:`
  or
  `remove:`

Naming Conventions

```
- (id) initWithBitmapDataPlanes: (char **)planes
            pixelsWide: (int)width
            pixelsHigh: (int)height
            bitsPerSample: (int)bps
            samplesPerPixel: (int)spp
            hasAlpha: (BOOL)alpha
            isPlanar: (BOOL)isPlanar
            colorSpaceName: (NSString *)colorSpace
            bytesPerRow: (int)rBytes
            bitsPerPixel: (int)pBits;
```

Naming Conventions

```
id image = [[NSBitmapImageRep alloc]
        initWithBitmapDataPlanes: planes
           pixelsWide: 32
           pixelsHigh: 32
           bitsPerSample: 32
           samplesPerPixel: 4
           hasAlpha: YES
           isPlanar: NO
           colorSpaceName: NSDeviceRGBColorSpace
           bytesPerRow: 0
           bitsPerPixel: 0];
```

Naming Conventions

```
NSBitmapImageRep image =
    new NSBitmapImageRep(
            planes,
            32,
            32,
            32,
            4,
            true,
            false,
            DeviceRGBColorSpace,
            0,
            0);
```

Naming Conventions

- Choose consistent terminology

Use

remove

rather than

delete

takeAway	**takeOut**
doAwayWith	**eliminate**
eradicate	**exterminate**
obliterate	**vaporize**

Naming Conventions

- Don’t abbreviate names in API

setFloatingPointFormat:	Good
setFloatingPntFormat:	Bad
setFltPtFmt:	Ugly

Naming Conventions

- If you do abbreviate, be consistent

```
alloc, allocWithZone:, dealloc
app
int
max, min
TIFF, RGB, USB, ASCII
```

Naming Conventions

- Avoid names that are ambiguous

```
sendPort
displayName
center
```

Naming Conventions

- Use verbs for methods which represent actions

```
selectCell:
removeObjectAtIndex:
```

Naming Conventions

- For methods that return values, use name of attribute or computed value being returned

```
- (NSSize) cellSize
```

not

```
- (NSSize) calculateCellSize
```

Naming Conventions

- Setters use set; getters don't use get

setColor:	color
setEditable:	isEditable
setDrawsBackground:	drawsBackground

Naming Conventions

- For return values by reference, use get

```
- (void) getRow: (int *)row  column: (int *)col
```

- Often used for multiple return values

API Conventions

- Use consistent types across APIs
    - Floats to represent coordinates instead of ints
    - NSPoint instead of two floats
    - NSString instead of char *
- Higher impedance matching across APIs

API Conventions

- nil is usually not a valid object argument
    - **appendString:**, **setTitle:**, etc. don’t accept nil
    - **subviews** returns empty array if no subviews
    - Can’t put nil in NSArray, NSDictionary
- But nil can be used to indicate runtime or other exceptional conditions

API Conventions

- Programming errors are indicated via exceptions (NSException) rather than error codes
    - Index out of bounds
    - Invalid nil arguments

Object Ownership

Passing Objects Around

- Object ownership is not transferred across calls

Memory Management Refresher

- Cocoa objects are reference counted
  - **[class alloc]**, **[class new]**, or **[obj copy]** create objects with reference count of one
  - **[obj retain]** adds a reference count
  - **[obj release]** removes a reference count
  - **[obj autorelease]** releases an object "later"
  - Objects are deallocated when their reference counts reach zero

Memory Management Refresher

- Cocoa objects are reference counted
  - **[class alloc]**, **[class new]**, or **[obj copy]** create objects with reference count of one
  - **[obj retain]** adds a reference count
  - **[obj release]** removes a reference count
  - **[obj autorelease]** releases an object “later”
  - Objects are deallocated when their reference counts reach zero

Passing Objects Around

- Object ownership is not transferred across calls
- When you pass someone an object
    - They will retain or copy it if they want
        - And release it when done
    - You don’t retain or copy it for them

Passing Objects Around

```
// Get the document’s title
NSString *string = [myDoc title];
// Capitalize it (“my picTUre” becomes “My Picture”)
NSString *capString = [string capitalizedString];
// Set the title to the capitalized string
[myDoc setTitle: capString];
```

Passing Objects Around

```
// Get the document's title
NSString *string = [myDoc title];
// Capitalize it ("my picTUre" becomes "My Picture")
NSString *capString = [string capitalizedString];
// Set the title to the capitalized string
[myDoc setTitle: capString];
```

Passing Objects Around

```
// Get the document’s title
NSString *string = [myDoc title];
// Capitalize it (“my picTUre” becomes “My Picture”)
NSString *capString = [string capitalizedString];
// Set the title to the capitalized string
[myDoc setTitle: capString];
```

Passing Objects Around

```
// Get the document’s title
NSString *string = [myDoc title];
// Capitalize it (“my picTUre” becomes “My Picture”)
NSString *capString = [string capitalizedString];
// Set the title to the capitalized string
[myDoc setTitle: capString];
```

Passing Objects Around

```
// Get the document’s title
NSString *string = [myDoc title];
// Capitalize it (“my picTUre” becomes “My Picture”)
NSString *capString = [string capitalizedString];
// Set the title to the capitalized string
[myDoc setTitle: capString];
```

```
// One line version: Capitalize the title of the document
[myDoc setTitle: [ [myDoc title] capitalizedString ] ];
```

Writing “Set” Methods

```
- (void) setTitle: (NSString *)newTitle {
   // Set the instance variable to the new title
   title = newTitle;
}
```

- This assumes “newTitle” will stick around

Writing “Set” Methods

```
- (void) setTitle: (NSString *)newTitle {
   // Make a copy of the new title
    title = [newTitle copy];
}
```

- This one copies the title
- But leaks the old value

Writing “Set” Methods

```
- (void) setTitle: (NSString *)newTitle {
    // First release the previous title
     [title release];
    // Now make a copy of the new title
     title = [newTitle copy];
}
```

- Better but not correct . . . .
- It might crash if title and newTitle are the same!

Writing “Set” Methods

```
- (void) setTitle: (NSString *)newTitle {
  if (title != newTitle) {
    // First release the previous title
    [title release];
    // Now make a copy of the new title
    title = [newTitle copy];
  }
}
```

- This one is also fine, and a little more efficient

Writing “Set” Methods

```
- (void) setTitle: (NSString *)newTitle {
    // Mark the previous title to be released later
    [title autorelease];
    // Now make a copy of the new title
    title = [newTitle copy];
}
```

- This is fine too

Which Is the Best?

```
- (void) setTitle: (NSString *)newTitle {
   if (title != newTitle)
      [title release];
      title = [newTitle copy];
   }
}
```

V.S.

```
- (void) setTitle: (NSString *)newTitle {
   [title autorelease];
   title = [newTitle copy];
}
```

- Do you **release** the old value, or **autorelease**?
- Answer depends on:
  - What the "get" method looks like
  - Usage pattern of the method

Writing “Get” Methods

```
- (NSString *) title {
    // Return the title instance variable
    return title;
}
```

- Simple and straightforward
- But, is it as safe as it can be?

Passing Objects Around (Redux)

```
// Get the document’s title
NSString *string = [myDoc title];
// Save the document
[myDoc save];
// Now attempt to use the title obtained below
[myLogFile recordAsSaved: string];
```

Passing Objects Around (Redux)

```
// Get the document’s title
NSString *string = [myDoc title];
// Save the document
[myDoc save];
// Now attempt to use the title obtained below
[myLogFile recordAsSaved: string];
```

Passing Objects Around (Redux)

```
// Get the document's title
NSString *string = [myDoc title];
// Save the document
[myDoc save];
// Now attempt to use the title obtained below
[myLogFile recordAsSaved: string];
```

Passing Objects Around (Redux)

```
// Get the document’s title
NSString *string = [myDoc title];
// Save the document
[myDoc save];
// Now attempt to use the title obtained below
[myLogFile recordAsSaved: string];
```

Passing Objects Around (Redux)

```
// Get the document's title
NSString *string = [myDoc title];
// Save the document
[myDoc save];
// Now attempt to use the title obtained below
[myLogFile recordAsSaved: string];
```

- Assumption is that the string is still valid
- What if the save operation changed the title, releasing the old one in the process?

Writing “Get” Methods

```
- (NSString *) title {
    return title;
}

- (void) setTitle: (NSString *)newTitle {
    if (title != newTitle) {
        [title release];
        title = [newTitle copy];
    }
}
```

Writing “Get” Methods

```
- (NSString *) title {
    return title;
}

- (void) setTitle: (NSString *)newTitle {
    if (title != newTitle) {
        [title release];
        title = [newTitle copy];
    }
}
```

Writing “Get” Methods

```
- (NSString *) title {
    return title;
}

- (void) setTitle: (NSString *)newTitle {
    if (title != newTitle) {
        [title release];
        title = [newTitle copy];
    }
}
```

- The above two methods violate the assumption

Writing “Get” Methods

```
- (NSString *) title {
   return [[title retain] autorelease];
}
- (void) setTitle: (NSString *)newTitle {
   if (title != newTitle) {
      [title release];
      title = [newTitle copy];
   }
}
```

- The above two methods together are fine

Writing “Get” Methods

```
- (NSString *) title {
    return title;
}
- (void) setTitle: (NSString *)newTitle {
    [title autorelease];
    title = [newTitle copy];
}
```

- The above two methods together are also fine

Writing “Get” Methods

```
- (NSString *) title {
    return title;
}
- (void) setTitle: (NSString *)newTitle {
    [title autorelease];
    title = [newTitle copy];
}
```

- Use this pair if “get” performance is important

Writing “Get” Methods

```
- (NSString *) title {
   return [[title retain] autorelease];
}
- (void) setTitle: (NSString *)newTitle {
   if (title != newTitle) {
      [title release];
      title = [newTitle copy];
   }
}
```

- Otherwise use this pair

Do You Copy or Retain?

```
- (void) setTitle: (NSString *)newTitle {
    [title autorelease];
    title = [newTitle copy];
}
```

Do You Copy or Retain?

```
- (void) setTitle: (NSString *)newTitle {
    [title autorelease];
    title = [newTitle retain];
}
```

- Copy makes a brand new copy of the object
- Retain increments reference count on the object
- Are you interested in the actual object itself, or the value of the object?

Do You Copy or Retain?

```
- (void) setTitle: (NSString *)newTitle {
    [title autorelease];
    title = [newTitle retain];
}
```

```
str = [NSMutableString string];    // Empty string
[str appendString: @“Hello”];      // Modify str
[doc1 setTitle: str];              // Set it as the title
[str appendString: @“World”];      // Modify str further
[doc2 setTitle:str];               // Set it as the title
```

Do You Copy or Retain?

```
- (void) setTitle: (NSString *)newTitle {
    [title autorelease];
    title = [newTitle retain];
}
```

```
str = [NSMutableString string];    // Empty string
[str appendString: @“Hello”];      // Modify str
[doc1 setTitle: str];              // Set it as the title
[str appendString: @“World”];      // Modify str further
[doc2 setTitle:str];               // Set it as the title
```

Do You Copy or Retain?

```
- (void) setTitle: (NSString *)newTitle {
    [title autorelease];
    title = [newTitle retain];
}
```

```
str = [NSMutableString string];    // Empty string
[str appendString: @“Hello”];      // Modify str
[doc1 setTitle: str];              // Set it as the title
[str appendString: @“World”];      // Modify str further
[doc2 setTitle:str];               // Set it as the title
```

Do You Copy or Retain?

```
- (void) setTitle: (NSString *)newTitle {
    [title autorelease];
    title = [newTitle retain];
}
```

```
str = [NSMutableString string];     // Empty string
[str appendString: @"Hello"];       // Modify str
[doc1 setTitle: str];               // Set it as the title
[str appendString: @"World"];       // Modify str further
[doc2 setTitle:str];                // Set it as the title
```

Do You Copy or Retain?

```
- (void) setTitle: (NSString *)newTitle {
    [title autorelease];
    title = [newTitle retain];
}
```

```
str = [NSMutableString string];     // Empty string
[str appendString: @“Hello”];       // Modify str
[doc1 setTitle: str];               // Set it as the title
[str appendString: @“World”];       // Modify str further
[doc2 setTitle:str];                // Set it as the title
```

Do You Copy or Retain?

```
- (void) setTitle: (NSString *)newTitle {
    [title autorelease];
    title = [newTitle retain];
}
```

```
str = [NSMutableString string];    // Empty string
[str appendString: @“Hello”];      // Modify str
[doc1 setTitle: str];              // Set it as the title
[str appendString: @“World”];      // Modify str further
```

This causes title of doc1 to change!

Do You Copy or Retain?

```
- (void) setTitle: (NSString *)newTitle {
   [title autorelease];
   title = [newTitle copy];
}
```

```
str = [NSMutableString string];    // Empty string
[str appendString: @“Hello”];      // Modify str
[doc1 setTitle: str];              // Set it as the title
[str appendString: @“World”];      // Modify str further
```

This feels better . . .

Neither Copy nor Retain

- Copy or retain imply ownership
- Can you keep an object without owning it?

```
- (void) setTarget: (id)obj {
    target = obj;
}
```

Neither Copy nor Retain

- Relationships which don’t imply ownership
  - NSView’s superview (parent)
  - NSControl’s target
  - NSTableView’s data source
  - Delegate
- OK to hold on to an object without retaining it
- Take care when releasing

Mutability

Mutability

- Mutable means Editable or Changeable
- Some objects are by nature only mutable
    - NSWindow
    - NSTableView
- Others, usually "value" objects, can exist as immutable objects
    - NSString
    - NSColor

NSString vs. NSMutableString

- NSString

```
- (NSString *) stringByAppendingString: (NSString *)s;

 s2 = [str stringByAppendingString: @“Hello”];
```

- NSMutableString

```
- (void) appendString: (NSString *)s;

 [str appendString: @“Hello”];
```

Why Have Immutable Objects?

- Performance
- Simpler implementation
- Thread safety
- Easier analysis of program logic

When to Use Immutable?

- Across API boundaries

```
- (void) setTitle: (NSString *)newTitle;
- (NSString *) title;
```

- In implementations

```
- (void) setTitle: (NSString *)newTitle {
    [title autorelease];
    title = [newTitle copy];
}
```

When to Use Mutable?

- In APIs, when it's important to expose the mutability
- Within a single block, to build an object incrementally

```
NSMutableString *str = [NSMutableString string];
for (cnt = 0; cnt < num; cnt++)
        [str appendString: [array objectAtIndex: cnt] ];
[myDoc setTitle: str];
```

Subclassing

Subclassing

- A very powerful object-oriented programming feature for creating “is-a” relationships
- In Cocoa, used relatively sparingly
    - Some classes are meant to be subclassed
    - And others can be subclassed
    - But usually aren’t

Subclassing Misuse

- Confusing “has-a” or “uses-a” relationship with “is-a”:

```
@interface Person : NSString {
    NSDate *birthDate;
    NSColor *eyeColor;
}
```

```
[aPerson isEqual:@“Joe”];
```

Subclassing Misuse

- Using the wrong “is-a” relationship:

```
@interface Person : Animal
```

Subclassing Misuse

- Not applying “is-a” correctly:

```
@interface Roommate : Pig
```

Inappropriate Subclassing

- When specializations of the subclasses need to change based on settings

```
@interface EditableTextField : Widget

@interface NoneditableTextField : Widget
```

Inappropriate Subclassing

- When the subclass changes the fundamental attributes of a class

```
@interface NSString : NSObject

@interface NSAttributedString : NSString
```

Inappropriate Subclassing

- NSString
  - Characters
  - Length
- NSAttributedString
  - Characters
  - Length
  - Attributes (text styles)

Inappropriate Subclassing

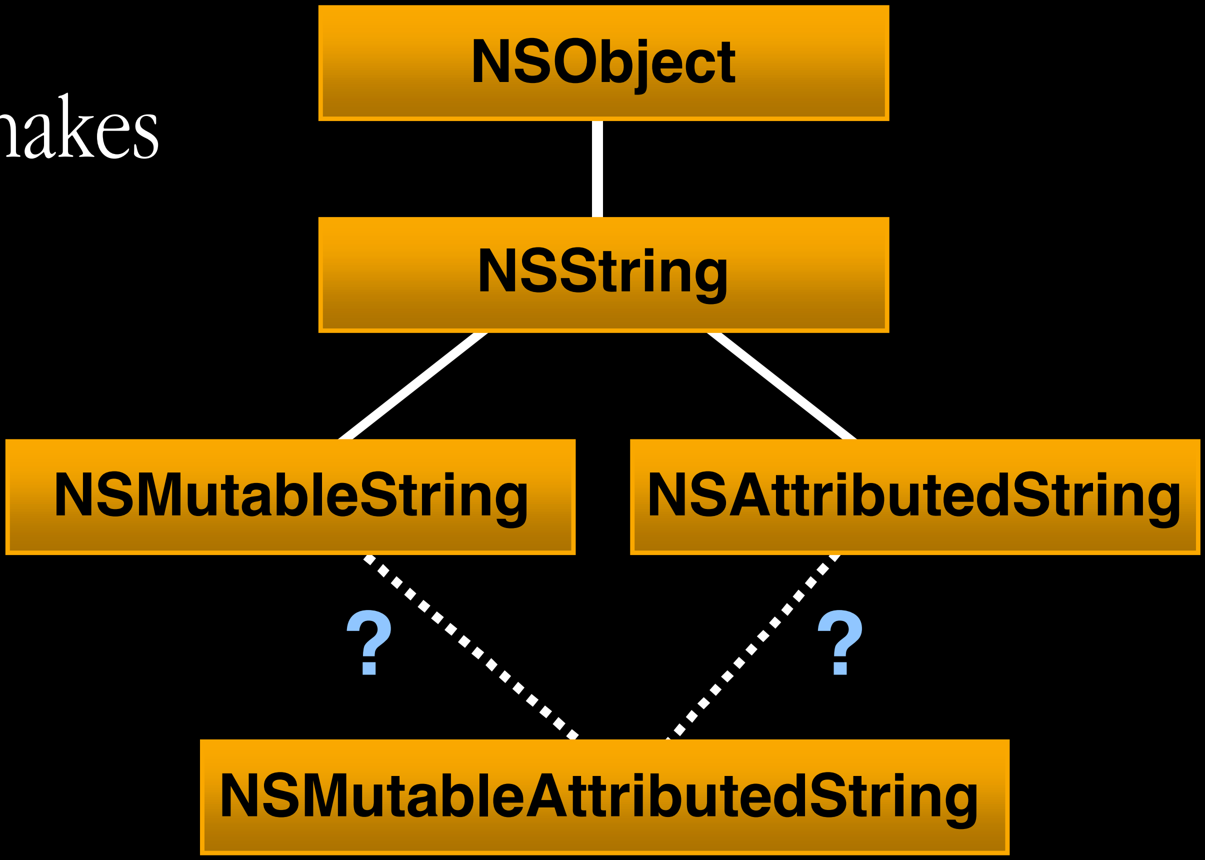
```
NSString *str;
NSAttributedString *attrStr;

// This might compare true
if ( [str isEqual: attrStr] ) { … }

// But this might not
if ( [attrStr isEqual: str] ) { … }
```

Inappropriate Subclassing

- No multiple inheritance makes subclassing awkward



Categories—An Alternative

- Categories allow additional method declarations and implementations on classes
    - All instances get the new methods
- Language feature

Categories

- Application Kit category on NSString:

```
@interface NSString (NSStringDrawing)

- (void) drawAtPoint: (NSPoint)point
         withAttributes: (NSDictionary *)attrs;

...
@end
```

Delegation—Another Alternative

- Allows an object to act on behalf of another
- Not a language feature
    - Classes explicitly support delegates
    - Each object has a single delegate

```
- (void) setDelegate: (id)obj;
- (id) delegate;
```

Delegation

- Some NSWindow delegate methods:

```
- (BOOL) windowShouldClose: (NSWindow *)w;

- (NSSize) windowWillResize: (NSWindow *)w
                     toSize: (NSSize)size;
```

Notification—Another Alternative

- Allows happenings to be broadcast to a set of unrelated observers
- Observers observe, but don't interfere
- Not a language feature
    - NSNotificationCenter provides the facility
    - Classes declare the notifications they post

Notification

- Some NSWindow notifications:
  NSWindowDidResizeNotification
  NSWindowWillCloseNotification

- Notifications are usually also sent to delegates:

```
- (void) windowDidResize: (NSNotification *)n;
- (void) windowWillClose: (NSNotification *)n;
```

Instead of Subclassing

- See if the class has delegate methods or notifications that will do what you want
- See if you can achieve the same results via category methods

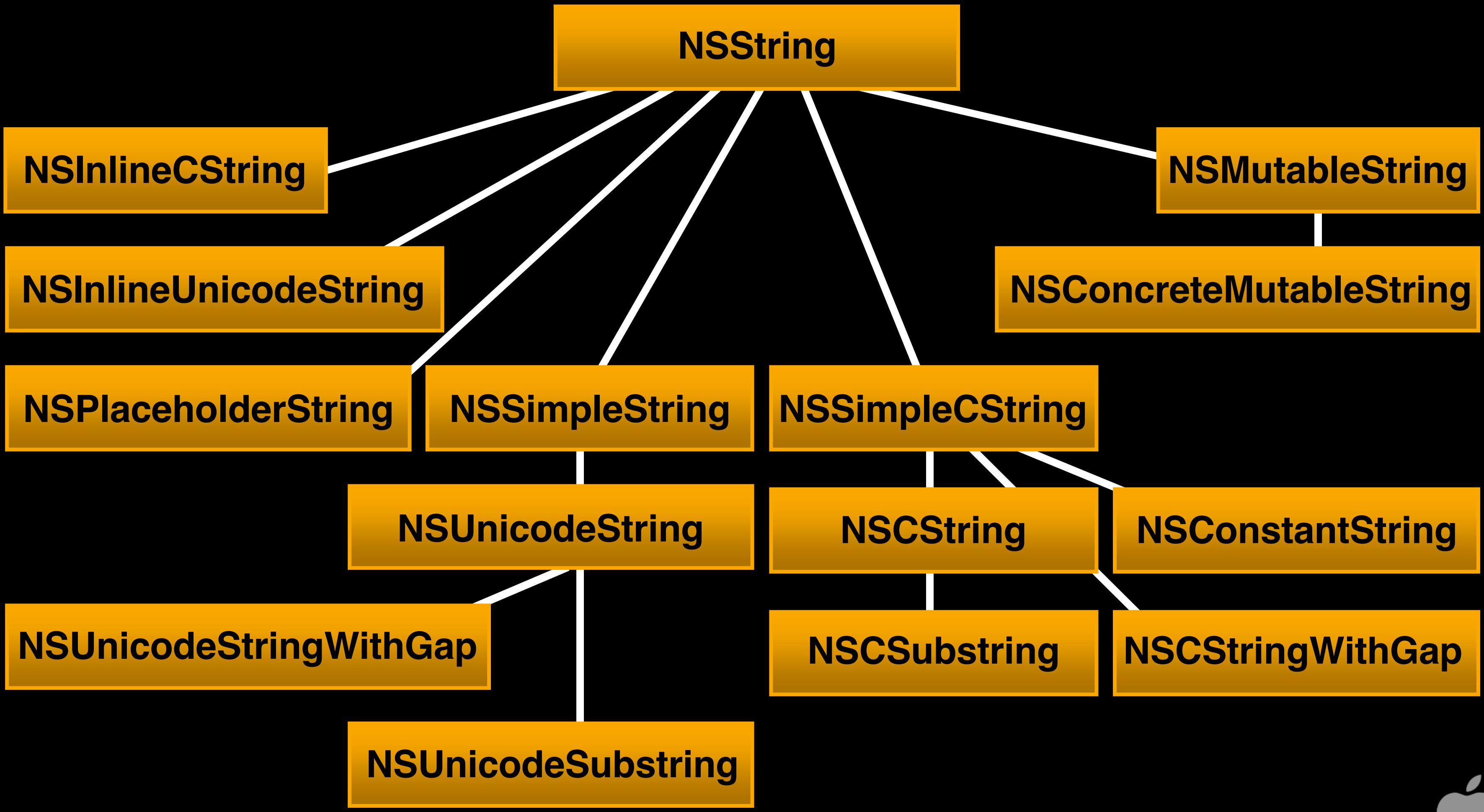
But If You Must Subclass

- Subclass!
- Know:
  - The superclass's primitives ("funnel" points)
  - Its designated initializers
  - And whether it's a class cluster

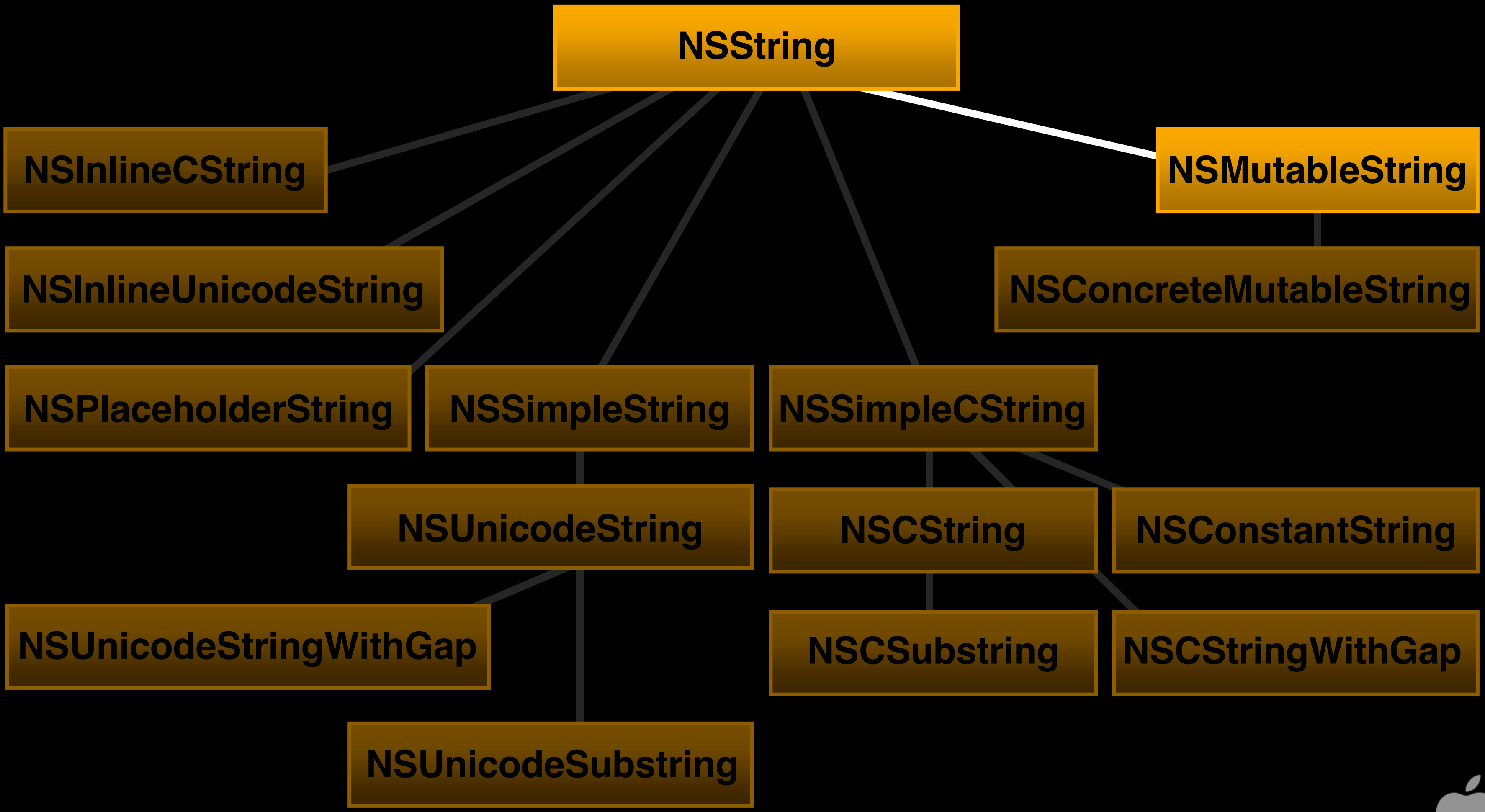
Class Clusters

NSString, A Few Years Ago

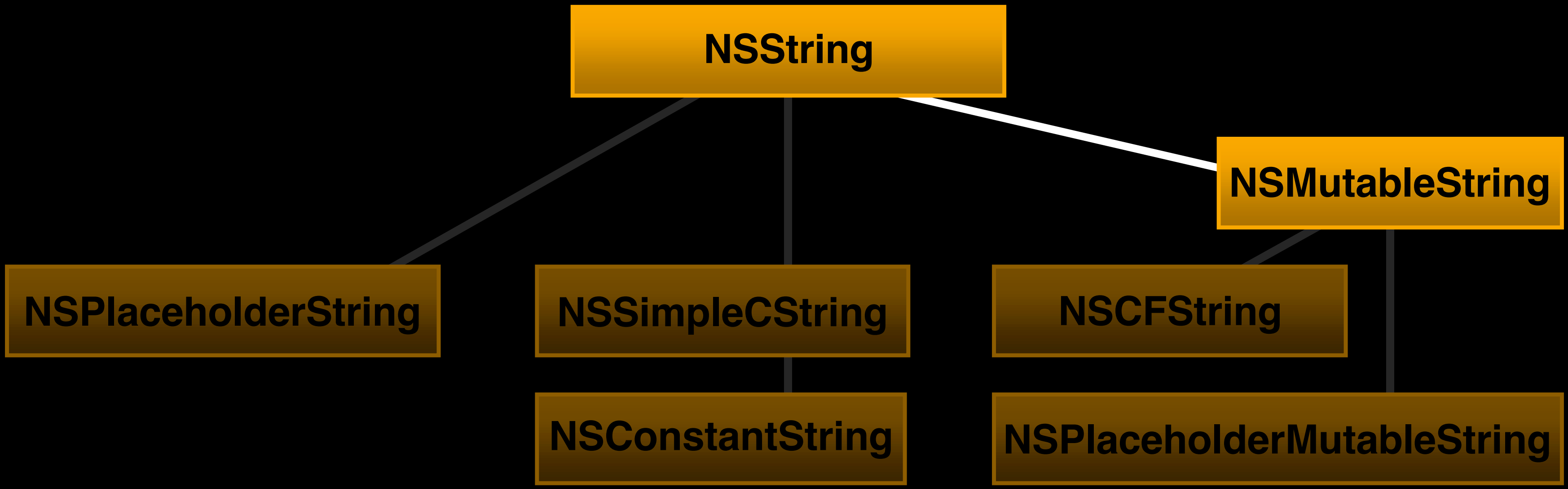
NSString, A Few Years Ago



NSString, A Few Years Ago



NSString, Today



Class Clusters

- Collection of implementation classes hidden and managed by an abstract superclass
- Subclassing usually requires a new implementation
- Primitives need to be clearly defined

Simple NSString Subclass

- Subclass to hold a “reversed” version of a string

```
@interface NSString : NSObject
- (unsigned) length;
- (unichar) characterAtIndex: (unsigned)index;
...
@end

@interface ReversedString : NSString {
   NSString *revStr;
 }
@end
```

Simple NSString Subclass

```
@implementation ReversedString : NSString
 - (id) initWithReverseOfString: (NSString *)str {
   if (self != [super init]) return nil;
   revStr = [str copy];    // revStr is the only instance variable
   return self;
}
 - (void) dealloc {
   [revStr release];
   [super dealloc];
}
 - (unsigned) length {
   return [revStr length];
}
 - (unichar) characterAtIndex: (unsigned)index {
   return [revStr characterAtIndex: [revStr length] - index - 1];
}
 @end
```

Simple NSString Subclass

```
@implementation ReversedString : NSString
 - (id) initWithReverseOfString: (NSString *)str {
    if (self != [super init]) return nil;
    revStr = [str copy];    // revStr is the only instance variable
    return self;
}
 - (void) dealloc {
    [revStr release];
    [super dealloc];
}
 - (unsigned) length {
    return [revStr length];
}
 - (unichar) characterAtIndex: (unsigned)index {
    return [revStr characterAtIndex: [revStr length] - index - 1];
}
 @end
```

Simple NSString Subclass

```
@implementation ReversedString : NSString
- (id) initWithReverseOfString: (NSString *)str {
   if (self != [super init]) return nil;
   revStr = [str copy];    // revStr is the only instance variable
   return self;
}
- (void) dealloc {
   [revStr release];
   [super dealloc];
}
- (unsigned) length {
   return [revStr length];
}
- (unichar) characterAtIndex: (unsigned)index {
   return [revStr characterAtIndex: [revStr length] - index - 1];
}
@end
```

Simple NSString Subclass

```
@implementation ReversedString : NSString
- (id) initWithReverseOfString: (NSString *)str {
   if (self != [super init]) return nil;
   revStr = [str copy];    // revStr is the only instance variable
   return self;
}
- (void) dealloc {
   [revStr release];
   [super dealloc];
}
- (unsigned) length {
   return [revStr length];
}
- (unichar) characterAtIndex: (unsigned)index {
   return [revStr characterAtIndex: [revStr length] - index - 1];
}
@end
```

Simple NSString Subclass

```
@implementation ReversedString : NSString
- (id) initWithReverseOfString: (NSString *)str {
    if (self != [super init]) return nil;
    revStr = [str copy];    // revStr is the only instance variable
    return self;
}
- (void) dealloc {
    [revStr release];
    [super dealloc];
}
- (unsigned) length {
    return [revStr length];
}
- (unichar) characterAtIndex: (unsigned)index {
    return [revStr characterAtIndex: [revStr length] - index - 1];
}
@end
```

Plug-in Design

Plug-in Design

- Package your plug-in as a bundle
- Use NSBundle to load it
    - Get the main class with
      `- (Class)principalClass`
    - Expose the entry points via the principal class

Exposing Entry Points

- Have a superclass meant to be overridden
- Supply the superclass implementation
  - ScreenSaver bundles
    - Subclasses of ScreenSaverView
  - Preferences panes
    - Subclasses of NSPreferencePane

Exposing Entry Points

- Use an Objective-C protocol
    - Protocols gather methods together with no implied class hierarchy
    - Classes conforming to a protocol implement all methods
- No base implementation needed

Exposing Entry Points

- Protocol declaration for a sample plug-in

```
@protocol ImageProcessingPlugIn <NSObject>
- (void) processImage: (NSImage *)img;
- (BOOL) canProcessImage: (NSImage *)img;
- (NSString *) description;
@end
```

Performance Tip

- Allow plug-in info to be gathered without loading the plug-ins
    - Provide the info in custom keys in Info.plist
    - Or in some other file in the bundle
    - Load the bundle and execute code lazily

Performance of APIs

Performance of APIs

- "As fast as possible" is a good goal
- But with many conflicting constraints
    - Reduce memory usage?
    - Optimize CPU usage?
    - Find a middle road?

Performance of APIs

- In many low-level APIs, it is important to indicate the performance characteristics
    - “20 nanoseconds per probe” is interesting
    - But behavior over a range of input parameters even more interesting

NSArray

- The performance of some NSArray APIs is not dependant on the number of objects in an array:

```
- (unsigned) count;
- (id) objectAtIndex: (unsigned)index;
- (void) addObject: (id)obj;
```

- But this is not true for all methods:

```
- (void) insertObject: (id)obj
              atIndex: (unsigned)index;
```

NSArray

- NSArray is not a linked list
- More like a C-style array
- If you create a linked-list, do not subclass NSArray

NSDictionary

- Stores key/value pairs
- Uses hashing
- Often the following execute in constant time:

```
- (id) objectForKey: (id)key;
- (void) setObject: (id)obj forKey: (id)key;
- (void) removeObjectForKey: (id)key;
```

NSDictionary

- Uses the following NSObject methods:

```
- (BOOL) isEqual: (id)obj;
- (unsigned) hash;
```

- Performance depends on:
  - Performance of these methods
  - Goodness of the hash function
- Worst-case performance of insertion and removal becomes dependent on the number of objects

NSDictionary

- Note:

```
[ obj1 isEqual: obj2 ]
```

implies

```
[ obj1 hash ]  ==  [ obj2 hash ]
```

NSString

- Has the primitive method:

```
- (unichar) characterAtIndex: (unsigned)i;
```

- Usually extremely fast, but invocation overhead can be a significant portion of the call
- Use the “bulk” method whenever possible:

```
- (void) getCharacters: (unichar *)buffer
                 range: (NSRange)range;
```

Conclusion

Cocoa Documentation

- Object-Oriented Programming and the Objective-C Language
- Programming Topics

Application Architecture	Memory Management
Foundation Framework	Multithreading
Loading Resources	Notifications

. . .and many more!

Documentation > Cocoa
developer.apple.com/techpubs/macosx/Cocoa/CocoaTopics.html

For More Information

- O'Reilly "Learning Cocoa" and "Building Cocoa Applications: A Step-by-Step Guide"
- Hillegass "Cocoa Programming for Mac OS X"
- Gamma, et al "Design Patterns"
- Cocoa Developer Documentation
  http://developer.apple.com/techpubs/macosx/Cocoa/CocoaTopics.html
- Apple Customer Training
  http://train.apple.com/

Roadmap

300 Introduction to Cocoa: What's Cocoa?	Room A1 **Mon., 5:00pm**
301 Cocoa: What's New: New features since Mac OS X 10.0	Civic **Tue., 9:00am**
303 Cocoa Scripting: Scripting overview and recent changes	Room A2 **Thurs., 10:30am**
304 Cocoa Controls & Accessibility: Overview of controls; new Accessibility APIs	Room A2 **Thurs., 5:00pm**
305 Cocoa Drawing: Drawing using Cocoa APIs	Hall 2 **Fri., 10:30am**

Roadmap

306 Cocoa Text: In depth overview of the text system	Room J **Fri., 2:00pm**
FF016 Cocoa: Comments and suggestions for Cocoa	Room A1 **Fri., 5:00pm**

Who to Contact

Heather Hickman
Cocoa Technology Manager
hhickman@apple.com

Cocoa Feedback
cocoa-feedback@group.apple.com

Cocoa Development Mailing List
Subscribe at
www.lists.apple.com/mailman/listinfo/cocoa-dev

Q&A

Heather Hickman
Cocoa Evangelist
hhickman@apple.com

http://developer.apple.com/wwdc2002/urls.html

WWDC2002

WWDC2002

WWDC2002